CSCI-3150: Introduction to Operating Systems

Lab Eight: Condition Variable

Deadline: April 28, 2021 23:59

1 Background

In this lab, you will learn how to use condition variables in pthread library to implement a wakeup-enabled multi-thread program. You will learn the following functions:

1. `pthread_cond_init()`: function to create a condition variable for wakeup-enabled access to shared resources.
2. `pthread_cond_wait()`: function to wait for a condition variable.
3. `pthread_cond_signal()`: function to wake up a waiting thread.

2 Creating Threads

The file `create_thread.c` is an example program of creating multiple threads. When a new thread is created, first we need to declare the function that the thread will execute. In the function `thr_func`, we ask the thread to print a message according to the data passed to the thread. The data is an integer indicating the thread id. Once the thread has finished its job, you need to call `pthread_exit` function to terminate the thread.

In the main program, we declare an array of `pthread_t`, the data structure storing thread information, and create 8 threads using `pthread_create()`. For each thread, we need to pass the address of `pthread_t` (also regarded as the thread handle), the thread function (need to be executed) and the data to this function. Once the thread is created, it will execute the code in the function `thr_func`. When `pthread_join()` is called, the main thread will be blocked until all the threads created in array `thr` have terminated.

Compile & Run: The program (`create_thread.c`) can be compiled and executed as follows:

```
$ gcc -o phello create_thread.c -lpthread
$ ./phello
```

Note: The compilation option `-lpthread` must be included when using pthread library.

From the result of the program, we can see that the main thread will print "All threads exited." only after all the thread has terminated.

```
hello from thr_func, thread id: 0
hello from thr_func, thread id: 1
hello from thr_func, thread id: 2
hello from thr_func, thread id: 3
hello from thr_func, thread id: 4
hello from thr_func, thread id: 5
```

```
hello from thr_func, thread id: 6
hello from thr_func, thread id: 7
All threads exited.
```

Note: Due to the indeterministic nature of thread execution (threads can be schedule to run by the OS in arbitrary order), the output of the child threads is not necessarily in a specific order. Other orders such as `7 6 5 4 3 2 1 0` are also possible.

Question: What if we try to remove the function call to `pthread_join()`?

3 Mutual Exclusion

It is common to share resources and data among multiple threads so that they can cooperate with each other to accomplish some tasks. However, we need to avoid multiple threads to access shared resources that cannot be accessed at the same time (e.g. critical section) since this may lead to unexpected behaviors. Mutual Exclusion is the method of serializing access to shared resources. In multi-threading programming, a common way to perform mutual exclusion is to use mutex.

The file `mutex.c` is an example program of resource sharing between multiple threads. The program is to create multiple threads, and each thread will add 10 to a shared variable `cnt`. In the file we declare a `pthread_mutex_t` variable named `mtx`, which is used to store the information of mutex. In the thread function, before the thread tries to add value to `cnt`, we need to call function `pthread_mutex_lock()` to lock the mutex `mtx`. If `mtx` is not locked by other threads, then the thread can lock it and enter the critical section. Otherwise, the thread will be blocked until the mutex is unlocked by other threads. After incrementing the number `cnt`, we call `pthread_mutex_unlock()` to unlock the mutex so that other threads could get access to the shared variable.

Compile & Run: The program (`mutex.c`) can be compiled and executed as follows:

```
$ gcc -o my-mutex mutex.c -lpthread
$ ./my-mutex
```

From the result of the program, it can be observed that each thread will increase the counter `cnt` when the mutex `mtx` is locked. 10 iterations will be repeated each thread, so after execution the `cnt` will become 80 (because we created 8 threads).

Question: What if we try to remove the function call to `pthread_mutex_lock()` and `pthread_mutex_unlock()`? What if the iterations are set to 10000? or 1000000?

4 Condition Variable Exercise

In this exercise, you are required to create some child threads and use condition variables to achieve the same behaviour as `pthread_join` and `pthread_exit`. The following program creates 5 (`N_THREADS`) and each of them prints a message before they call `thr_exit`. The main function calls `thr_join` to wait for the threads to terminate before it returns. You should fill in all `/*YOUR CODE HERE */` with your own code to complete the program.

```
#include <stdio.h>
#include <pthread.h>
```

```c

#define N_THREADS 5

static int exited;

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void thr_exit() {
  pthread_mutex_lock(&mutex);

  /* YOUR CODE HERE */

  pthread_mutex_unlock(&mutex);
}

void thr_join() {
  pthread_mutex_lock(&mutex);

  while (exited < N_THREADS) /* YOUR CODE HERE */

  pthread_mutex_unlock(&mutex);
}

void *child_func(void *arg) {
  int thr_id = /* YOUR CODE HERE */;
  printf("child %d created and exited\n", thr_id);
  thr_exit();
  return NULL;
}

int main() {
  pthread_t p[N_THREADS];
  int thr_idx[N_THREADS];
  void *arg;
  int i;

  exited = /* YOUR CODE HERE */;

  puts("parent: begin");

  for (i = 0; i < N_THREADS; i++) {
    thr_idx[i] = i;
    arg = &thr_idx[i];

```

```
    pthread_create(/* YOUR CODE HERE */);
  }

  thr_join();

  puts("parent: end");

  return 0;
}
```

You can find the program code in `exercise.c`. To compile and run the program, use:

```
$ gcc -o exercise exercise.c -lpthread
$ ./exercise
```

Sample output:

```
parent: begin
child 0 created and exited
child 3 created and exited
child 1 created and exited
child 4 created and exited
child 2 created and exited
parent: end
```

Note: Because the threads are scheduled to run in arbitrary order, the order of child output can be different from the sample output. **If you correctly implement `thr_join` and `thr_exit`, you should see `"parent: end"` at the end of the output, after all child outputs**. Also, it is possible that all child threads run and finish before the main thread call `thr_join` so you may get the sample output even if the two functions are not correctly implemented. You should run the program multiple times to detect potential bugs.

5 Submission

You ONLY need to submit the completed exercise.c to Blackboard. If you have any questions about this assignment, please send a email to jinxue@cse.cuhk.edu.hk.